Final Project - Machine Learning Solutions

Enrique Chi Gongora, *1909040*, Jose Pat Ramírez, *2009104*, Rafael Marí Reyna, *2009083*,
Jesus Gabriel Canul, *1909031*, Hernando Te Bencomo, *2009132*, Raymundo Baas Cabañas, *1909008*,
Victor Alejandro Ortiz Santiago.

***Abstract*—This project presents three distinct machine learning approaches to address practical problems in robotics: supervised learning, unsupervised learning, and reinforcement learning. The first part employs supervised learning for object detection using sensor data, focusing on accuracy and model improvement through diverse datasets. The second part explores unsupervised learning via k-means clustering to analyze robot navigation patterns, employing PCA for data dimensionality reduction. The final section introduces a reinforcement learning solution, where a custom environment, training regime, and agent are developed to enable a robot to find the fastest route in a simulated environment. This section utilizes the Gym library to create a realistic and interactive environment, training the agent to make optimal decisions through trial and error. Each method demonstrates unique strengths and challenges in applying machine learning to robotic systems, highlighting the potential and versatility of these approaches in solving complex real-world problems.**

I. Objectives

A. Supervised Learning

- Develop a model capable of accurately detecting objects using sensor data in the context of robotics and automation.
- Train the model using simulated ultrasonic sensor data to ensure effective operation in various environmental conditions.
- Enhance the model's performance through data diversification, hyperparameter tuning, and advanced feature engineering techniques.

B. Unsupervised Learning

- Employ unsupervised learning, specifically k-means clustering, to analyze robot navigation patterns.
- Utilize sensor data from a wall-following robot to identify clusters representing different navigation behaviors.
- Explore the use of unsupervised learning to extract meaningful insights from unlabelled data, focusing on robotic navigation improvement.

C. Reinforcement Learning

- Develop a reinforcement learning system for a robot to learn the fastest route in a simulated environment.
- Create a realistic and interactive environment using the Gym library for the robot to navigate.
- Train the agent to optimize its decision-making process through trial and error, adapting to dynamic scenarios.

II. Introduction

The project explores three distinct machine learning techniques in the context of robotics: supervised learning, unsupervised learning, and reinforcement learning. Each section addresses a specific challenge in robotics, using a different learning approach.

The first part of the project focuses on object detection through supervised learning, which represents a dynamic and challenging field in artificial intelligence, with applications transforming robotics and automation. The project addresses this challenge by developing a machine learning model capable of identifying the presence of objects using simulated data from ultrasonic sensors. With an approach to data generation, algorithm simulates realistic and complex scenarios, taking into account variables such as position, energy consumed and environmental conditions. The resulting dataset is diverse and contains multiple dimensions of information, creating a strong basis for training a model capable of detecting subtle variations in objects. Through pre-processing and the application of modeling techniques, we aim to create a versatile system for integration into robotics applications, contributing to the improvement of machine-environment interaction on multiple platforms.

To understand how the project was developed, let's understand the main part of it. Supervised learning is a fundamental approach in machine learning, where the goal is to teach computers how to perform tasks based on examples. The computer uses the data provided to learn how to map inputs to outputs, for example. The process of supervised learning has two main phases: training and testing. During training, the algorithm is exposed to a large dataset, learning to understand the relationship between the input data and their corresponding labels. In testing, the algorithm is given new data to see how well it can apply what it learned to make accurate predictions.

The second part of our project uses unsupervised learning to analyze robot navigation. Our project employs unsupervised learning, particularly k-means clustering, to work on the datasets gathered from a wall-following robot equipped with ultrasound sensors. This approach helps us understand the robot's navigation patterns and identify areas for improvement. The essence of this method lies in its ability to identify structures and behaviors in the navigation data that are not immediately apparent.

By applying techniques such as Principal Component Analysis (PCA) for dimensionality reduction, we enhance our ability to interpret the data effectively. By examining the

robot's data, we aim to gain insights into its movement and explore strategies for better navigation, leading to more efficient and intelligent autonomous movement. This part of our project focuses on the capabilities of unsupervised learning for extracting meaningful information from unstructured data.

Finally, we have reinforcement learning. The reinforcement learning (RL) section of our project introduces a dynamic and interactive approach, focusing on training a robot to find the most efficient route in a simulated environment. Here, we create a scenario where a robot learns to navigate through trial and error. The robot experiments with various movements, learning from the outcomes to improve its pathfinding strategies and acting as an agent that learns from its interactions within the environment.

The core of reinforcement learning focuses on the agent's ability to make decisions, receive feedback in the form of rewards or penalties, and use this feedback to adjust its actions. This process mimics the way humans and animals learn, making it an effective approach for teaching robots how to adapt to new challenges. As the robot explores differents strategies and learns from the outcomes, it gradually improves its navigation skills, demonstrating the adaptability and potential of reinforcement learning in real-world scenarios.

Overall, our project aims to show how different machine learning methods can be applied to solve various challenges in robotics, showcasing the potential and versatility of machine learning in this field.

III. Methods and Tools

A. Supervised Learning

The methodology employed in this project involves a multifaceted approach to data generation and processing, as well as the selection and training of the most appropriate machine learning model. Initially, sensory data is simulated through specialized Python functions, such as simulate_ultrasonic_reading, which creates realistic distance readings based on the relative position and presence of objects. This simulation is further enhanced by simulate_motion and simulate_other_features, which add motion dynamics and other variables such as velocity and ambient temperature. These functions are explained in detail in the appendices of the document.

The simulated data are stored in a CSV file, and a copy of this is worked with to preserve the integrity of the original data set. Data cleaning and transformation of categorical variables into numerical variables is performed using the pandas library. Subsequently, the dataset is divided into training and test subsets using the scikit-learn tool.

B. Unsupervised Learning

The dataset was retrieved from the following link: Wall-Following Robot Navigation Data, where where data collection was conducted using a SCITOS G5 robot. This robot navigated a room's perimeter in a clockwise direction four times, equipped with 24 circularly arranged ultrasound sensors around its midsection.

The provided files include three different datasets. The first one presents raw sensor readings from all 24 ultrasound sensors, along with their respective class label. These readings were taken at a frequency of 9 samples per second.

The second one contains four sensor readings named 'simplified distances' and the corresponding class label. These simplified distances are referred to as the 'front distance', 'left distance', 'right distance' and 'back distance'. These distances are the minimum sensor readings among those within 60 degree arcs located at each respective orientation of the robot.

The third one contains only the front and left 'simplified distances' and their associated class labels. It is worth noting that the 24 ultrasound readings and the simplified distances were collected at the same time step, so each dataset has an equivalent number of rows (one for each sampling time step).

The purpose of the wall-following task and data collection was to test the hypothesis that this navigation task is actually a non-linearly separable classification task. Thus, linear classifiers, such as the Perceptron, are not able to learn this task and control the robot around the room with no collisions. However, non-linear neural classifiers, such as the MLP network, can learn the task and navigate the robot safely and without collisions.

If short-term memory mechanisms are incorporated into neural classifiers, their overall performance generally improves. For example, if past inputs are provided together with current sensor readings, even the Perceptron becomes able to learn and succeed in the task. On the other hand, if a recurrent neural network, such as Elman, is used to learn the task, it can do so using even fewer hidden neurons than an MLP network.

Datasets with different number of sensor readings were compiled to evaluate how the number of inputs affects the performance of classifiers.

C. Reinforcement Learning

The reinforcement learning component of the project was developed using a combination of Python libraries and custom scripts. The primary tool used for creating the learning environment is the OpenAI Gym library, a toolkit for developing and comparing reinforcement learning algorithms.

The custom environment within this solution, whise is named 'FastestRouteFindingRobotEnv', is designed to simulate a robot navigating through a space to reach a goal. This environment is built using Gym's environment class, and it includes custom definitions for the state and action spaces. The state space represents the possible positions of the robot in the

environment, while the action space determines the potential actions the robot can take, like moving in different directions.

The training script focuses on the learning process. It involves initializing the environment, running episodes where the agent (the robot) interacts with the environment, and updating its behavior based on the rewards received. The agent's learning algorithm, detailed in agent.py, is responsible for deciding actions at each step and improving these decisions over time through trial and error. This script uses NumPy for numerical computations, which are crucial for the decision-making process.

This part of the project demonstrates the practical application of reinforcement learning in robotics, showcasing how an agent can autonomously develop sophisticated navigation skills through interaction and feedback. The methodology employed demonstrates the potential of reinforcement learning in enabling robots to adapt and learn in dynamic environments.

IV. Development

A. Supervised Learning

After a comparative evaluation of several models, the Support Vector Machine (SVM) model was chosen for its higher performance. SVM is a powerful and versatile supervised learning algorithm whose main idea is to find a way to separate different categories of data points with a clear gap that's as wide as possible, using what is called in machine learning a hyperplane. In two dimensions, this hyperplane is a line, but in more complex datasets with many features, it can be a multi-dimensional surface.

When it comes to tuning an SVM, there are a couple of key parameters that need careful adjustment. The most important are the choice of the kernel function and the regularization parameter. The kernel function defines the type of hyperplane and transformation used for separation, while the regularization parameter controls the trade-off between having a wide margin and correctly classifying training data. A higher value of this hyperparameter tries to classify all training examples correctly, which can lead to overfitting, while a lower value results in a wider margin, possibly with some misclassifications in the training set.

For those reasons, these hyperparameters such as the kernel and the CC regularization parameter were optimized, seeking to improve the accuracy of the model. Several tests were performed, the results of which were measured with accuracy metrics and visualized through confusion matrices.

For the visualization of the results, graphical libraries such as matplotlib and seaborn were used. Seaborn is a Python data visualization library based on Matplotlib that offers a high-level interface for drawing attractive and informative statistical graphics. It is particularly suited for making complex plots more accessible. The robot trajectory was illustrated along with object detection predictions, providing a clear graphical representation of the model's ability to identify obstacles. The accuracy of the model and its interpretation were significantly improved by hyperparameter tuning of the SVM, which was identified as the most effective model among those tested. The entire process from data generation to visualization of the results is documented in the appendices to provide a complete guide to the methodological approach adopted in the project.

B. Unsupervised Learning

By solving the separator issue, the file was successfully loaded using the pandas 'read_csv' function, explicitly specifying the correct comma separator. The data was organized into a DataFrame, and numerical features representing sensor readings were extracted for further analysis.

Additionally, class labels were separated for subsequent visualization purposes. The extracted numerical features underwent standardization using the 'StandardScaler' from scikit-learn, a crucial preprocessing step to ensure all features had a similar scale, facilitating accurate clustering. Subsequently, K-means clustering was applied to group similar data points into clusters, with the number of clusters (k) set to 4 in this example, a value subject to adjustment based on the desired level of granularity in the analysis.

Principal Component Analysis (PCA) was employed for dimensionality reduction, reducing the feature space to 2 principal components for visualization. The clustered data points were plotted in a 2D space, providing insights into the patterns and relationships among the sensor readings. The use of PCA aids in visually understanding the structure of the data and the effectiveness of the clustering algorithm.

The class labels, initially separated from the numerical features, were visualized separately to provide additional context to the analysis. Class labels were plotted in the same reduced feature space to observe how the clusters align with different behaviors of the robot. This dual visualization enhances the interpretability of the clustering results.

Moving forward, there is room for improvement and several ways to do so. Exploring hyperparameter tuning, particularly experimenting with different values of 'k', can help identify the optimal configuration for the dataset. Consideration of advanced clustering algorithms beyond k-means, such as hierarchical clustering or density-based clustering, may result in additional insights or patterns. Feature engineering should be explored to enhance the discriminatory power of the clustering model, especially in datasets with varied patterns like those in robotics.

Integration of additional information, such as temporal aspects or correlations between sensor readings, can be considered for a more comprehensive model. Enhanced visualization techniques, including 3D plots or interactive visualizations, can provide a richer understanding of the data structure and clustering results.

Lastly, the implementation of quantitative metrics, such as silhouette score or Davies-Bouldin index, can offer objective measures of cluster quality for thorough model evaluation. By addressing these future improvements, the analysis can be

refined to uncover more nuanced patterns and enhance the overall quality of the unsupervised learning approach applied to the Wall-Following Robot Navigation dataset.

C. Reinforcement Learning

The reinforcement learning part of the project involved developing and training an agent to find the fastest route in a simulated environment. This process was carried out using the custom FastestRouteFindingRobotEnv environment, created with the OpenAI Gym library, and a reinforcement learning algorithm implemented in Python.

The environment setup in environment.py defined a simple, yet challenging space for the agent to navigate. The state space represented the robot's position, and the action space consisted of discrete actions the robot could take to move within the environment. The goal was to reach a specific target position in the shortest number of steps.

With the agent's learning algorithm, as defined in agent.py, it decided the actions based on the current state. This was improved iteratively through the training process. The training script in train.py managed the interaction between the agent and the environment, running multiple times to gather data and refine the agent's learning.

During training, the agent received rewards for each action, with higher rewards for actions that moved it closer to the goal. The reinforcement learning algorithm used these rewards to evaluate the effectiveness of actions and adjust the policy accordingly. This process of trial and error allowed the agent to learn the most efficient path to the goal over time.

After the training was complete, the performance of the agent was evaluated by running it in the environment and measuring the efficiency of its navigation skills. Future improvements in this area could involve experimenting with different reinforcement learning algorithms, adjusting the complexity of the environment, or introducing more sophisticated reward systems. These changes could enhance the agent's ability to learn and adapt, potentially leading to more efficient and robust navigation strategies.

V. Results

A. Supervised Learning

The results of the object detection model using a Support Vector Machine (SVM) indicate moderate performance with an accuracy of 56%. This accuracy metric is calculated as the number of correct predictions divided by the total predictions made. An accuracy of 56% suggests that the model is slightly better than a random choice, which would have an expected accuracy of 50% in a binary classification problem such as this, but there is certainly room for improvement.

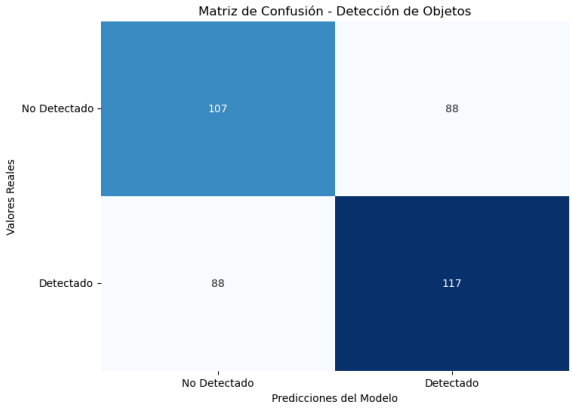

Fig. 1. Confusion Matrix

The matrix shows that the model correctly predicted the absence of an object (True Negatives) 107 times and the presence of an object (True Positives) 117 times. However, there were 88 False Positives, where the model erroneously predicted the presence of an object, and 88 False Negatives, where the model failed to detect an object that was present.

Visually, this is reflected in the graph of the robot trajectory with obstacle predictions. The points marked with a red 'x' represent locations where the model predicted the presence of an obstacle, and the blue line shows the robot's trajectory. The scatter of red 'x's relative to the blue trajectory provides a visual intuition of where and how many times the model perceived obstacles.

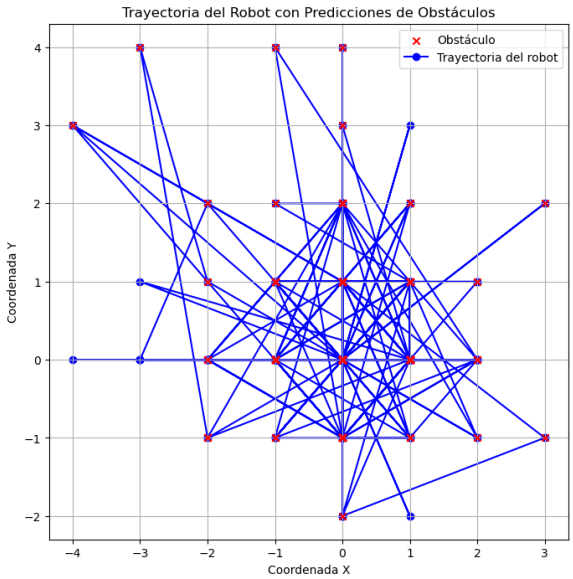

Fig. 2. Trajectory simulation and object detection.

The accuracy of the model is a starting point, but for practical applications, especially in robotics where wrong decisions can have significant consequences, it would be vital to improve this model. This could include collecting more data, testing different types of models, tuning hyperparameters, or employing feature engineering techniques to improve the model's ability to generalize from training data and make more accurate predictions on unseen data.

B. Unsupervised Learning

The application of K-Means clustering to the standardized numerical features of the wall-following robot navigation dataset revealed distinct patterns, as illustrated in the scatter plot. With four clusters, the unsupervised learning approach grouped data points, offering insights into potential underlying structures. Subsequent Principal Component Analysis (PCA) further visualized the dataset in a reduced two-dimensional space, providing a clearer understanding of relationships and separations among clusters. While these techniques provide valuable insights, further exploration and parameter tuning could enhance the model's interpretability and overall performance.

It's crucial to interpret the clustering results within the context of the specific problem domain. The analysis serves as a starting point for understanding the dataset's structure, but refinement and feature engineering may be necessary for more nuanced insights. Overall, the combined use of K-Means clustering and PCA offers an effective means of exploring and visualizing patterns in the wall-following robot navigation dataset, guiding further investigation and potential improvements in model understanding.

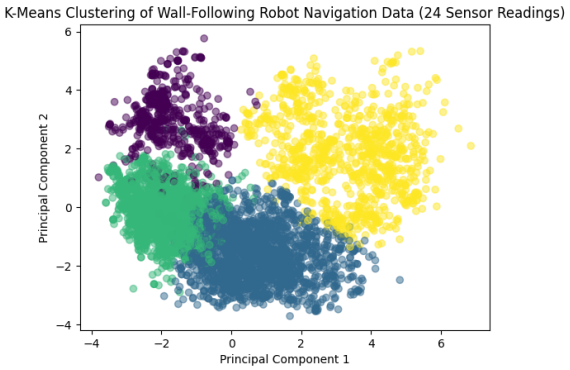

Fig. 3. Clustering representation of the repetitive most movements.

The matrix shows that the model correctly predicted the absence of an object (True Negatives) 107 times and the presence of an object (True Positives) 117 times. However, there were 88 False Positives, where the model erroneously predicted the presence of an object, and 88 False Negatives, where the model failed to detect an object that was present.

Visually, this is reflected in the graph of the robot trajectory with obstacle predictions. The points marked with a red 'x' represent locations where the model predicted the presence of an obstacle, and the blue line shows the robot's trajectory. The scatter of red 'x's relative to the blue trajectory provides a visual intuition of where and how many times the model perceived obstacles.

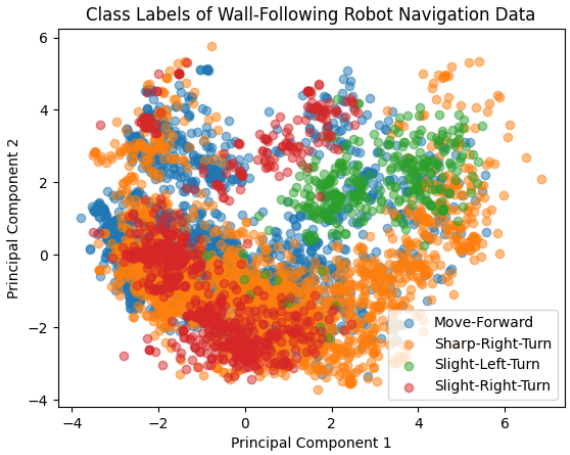

Fig. 4. Trajectory Simulation

The accuracy of the model is a starting point, but for practical applications, especially in robotics where wrong decisions can have significant consequences, it would be vital to improve this model. This could include collecting more data, testing different types of models, tuning hyperparameters, or employing feature engineering techniques to improve the model's ability to generalize from training data and make more accurate predictions on unseen data.

C. Reinforcement Learning

In the domain of machine learning, and more specifically reinforcement learning, the code in question illustrates the training of an agent using Q-Learning, a tabular method that allows the agent to learn through interaction with its environment. The simulated environment is one-dimensional, where the agent, a robot in this case, must move from a starting point to a target point. The Q-Learning methodology employs a Q-table to record and update values that reflect the expected utility of performing certain actions in certain states. Initially, the agent's strategy is exploratory, opting for random actions that allow it to acquire knowledge about the environment. Over time, and as it accumulates experience, the agent gradually reduces this random exploration by a decay factor, moving toward a policy of exploiting the Q-table to select the best action based on previous rewards and learned values. The agent is rewarded for reaching its goal and penalized based on the distance to the goal, thus incentivizing efficiency in the navigation task. After a thousand training episodes, the agent shows a consistent ability to reach the target, which is evidenced by the stable reward of 1.0 obtained at the test intervals. This signals that the agent has achieved an effective policy that balances the initial exploration necessary to understand the environment and the subsequent exploitation of its accumulated knowledge to maximize the overall reward, a desired outcome in reinforcement learning.

```
Episode 0, Exploration Probability: 0.499, Steps: 14, Reward: 1.0
Episode 100, Exploration Probability: 0.3989999999999999, Steps: 14, Reward: 1.0
Episode 200, Exploration Probability: 0.2989999999999998, Steps: 14, Reward: 1.0
Episode 300, Exploration Probability: 0.19899999999999973, Steps: 14, Reward: 1.0
Episode 400, Exploration Probability: 0.1, Steps: 14, Reward: 1.0
Episode 500, Exploration Probability: 0.1, Steps: 14, Reward: 1.0
Episode 600, Exploration Probability: 0.1, Steps: 14, Reward: 1.0
Episode 700, Exploration Probability: 0.1, Steps: 14, Reward: 1.0
Episode 800, Exploration Probability: 0.1, Steps: 14, Reward: 1.0
Episode 900, Exploration Probability: 0.1, Steps: 14, Reward: 1.0
```

Fig. 5. Evolution of exploration and rewards in training

VI. Discussion

The outcomes of the project, which includes supervised, unsupervised, and reinforcement learning approaches, reveal different challenges and opportunities for improvement in the application of machine learning in robotics.

In the supervised learning part, discussion of the results indicates that the model has a level of accuracy that requires improvement to be effective in practical applications. With an accuracy of 56%, the model does not demonstrate sufficient reliability for real-world applications where accurate distinction between the presence and absence of objects is critical. The confusion matrix shows a considerable number of errors in both object detection and non-detection, implying that the model is often wrong in predicting.

These results are important because they point to areas where the project could be developed further. For example, the data being used to train the model could be reviewed and adjusted, or the use of different methods to build the model that might work better with this type of data could be explored. The discussion focuses on understanding how well the current model works and what steps could be taken to make it more accurate and reliable in the future.

In the case of unsupervised learning, the issue obtained from the data in the file was not being recognized correctly due to an incorrect separator, causing a failure in converting a string to a float during the standardization process. The initial attempt to load the data assumed a default separator, leading to the inclusion of class labels in the last column as a single string. To address this, a modification was made to explicitly specify the correct separator, which, in this case, was a comma. This adjustment in the 'pd.read_csv' function ensured that the data was properly separated into individual values, resolving the issue with reading and processing the dataset. The successful execution of the code following this modification highlighted the importance of accurately specifying data separators, particularly in real-world scenarios where datasets may exhibit diverse characteristics.

Regarding the reinforcement learning part, the project developed an agent capable of navigating a simulated environment, demonstrating the potential of reinforcement learning in dynamic learning and decision-making. However, the agent's performance highlighted the importance of having a balance between exploration and exploitation, a fundamental aspect of reinforcement learning. Optimizing this balance, possibly through parameter adjustment or the introduction of more sophisticated learning algorithms, could significantly enhance the agent's navigation efficiency. Further exploration could include making the environment more complex or refining the reward system to challenge the agent and improve its adaptability to more difficult scenarios.

VII. Conclusion

In this project, three types of machine learning methods were examined for their use in robotics: supervised learning, unsupervised learning, and reinforcement learning. Each method provided unique insights into the learning and decision-making processes of robots.

In the supervised learning part, a model for object detection was developed. It was found that the model's accuracy needs to be improved for real-world applications. This suggests that further work is required, such as trying different training methods or using more varied data. The need for highly accurate object detection in robotics makes this an important area for ongoing development.

The project's unsupervised learning section focused on understanding robot movement. Initially, there were issues with processing the data, but once these were resolved, valuable insights were gained. This part of the project indicates that exploring different data grouping techniques or adding more detailed features could provide deeper understanding.

For reinforcement learning, a robot was trained in a simulated environment. It was seen that the robot improved over time, but the process of how it learns to navigate efficiently remains a key challenge. Future efforts might look at using more complex scenarios or improving the robot's learning approach.

Overall, the project showed how each machine learning method can be useful in robotics, but also where improvements are needed. The findings point towards future work that can help make robots better at learning and adapting in various situations.

VIII. References

1) N. Bellotto, K. Burn, and S. Wermter, "Appearance-based localization for mobile robots using digital zoom and visual compass," *Robot. Auton. Syst.*, vol. 56, pp. 143–156, 2008. DOI: https://doi.org/10.1016/j.robot.2007.09.004
2) K. Y. Chan, R. Manduchi, and J. Coughlan, "Accessible spaces: Navigating through a marked environment with a camera phone," in *Proceedings of the 9th International ACM SIGACCESS Conference on Computers and Accessibility*, Tempe, AZ, USA, 14–17 October 2007.
3) "DOF a entidades de obstáculo (Inteligencia)—ArcGIS Pro — Documentación," Accedido el 13 de noviembre de 2023. En línea. Disponible: https://pro.arcgis.com/es/pro-app/3.0/tool-reference/intelligence/dof-to-obstacle-features.htm
4) "Obstacle Avoidance Robot Object Detection Dataset (v1, 2023-07-13 2:05pm) by Moratuwa Project," Roboflow. Accedido el 13 de noviembre de 2023. En línea. Disponible: https://universe.roboflow.com/moratuwa-project/obstacle-avoidance-robot/dataset/1
5) E. Yi Kim, "Wheelchair Navigation System for Disabled and Elderly People," *empirica*, 2016.
6) A. Freire, M. Veloso y G. Barreto. "UCI Machine Learning Repository". UCI Machine Learning Repository. Accedido el 5 de diciembre de 2023. [En línea]. Disponible:

https://archive.ics.uci.edu/dataset/194/wall+following+robot+data https://doi.org/10.24432/C57C8W

APPENDIX A
SUPERVISED LEARNING - SIMULATED DATA GENERATION

```
def simular_lectura_ultrasonico(posicion,
    objeto_presente):

    if 'contador_lecturas' not in
        simular_lectura_ultrasonico.__dict__:
        simular_lectura_ultrasonico.contador_lecturas =
            0

    if objeto_presente and
        simular_lectura_ultrasonico.contador_lecturas <
        40:
        simular_lectura_ultrasonico.contador_lecturas +=
            1
        distancia_base =
            np.linalg.norm(posicion)
        variabilidad =
            random.uniform(0, 1) * 5
        distancia_objeto =
            distancia_base +
            variabilidad
    else:
        distancia_objeto = 0

    return distancia_objeto
simular_lectura_ultrasonico.contador_lecturas =
    0

def simular_movimiento():
    direccion = random.uniform(0, 2 *
        np.pi)
    distancia = random.uniform(0, 2)
    movimiento = {'direccion':
        direccion, 'distancia':
        distancia}
    return movimiento

def simular_otras_caracteristicas():
    velocidad_lineal =
        random.uniform(0, 1)
    velocidad_angular =
        random.uniform(0, 1)
    aceleracion_lineal =
        random.uniform(0, 1)
    aceleracion_angular =
        random.uniform(0, 1)
    tiempo = random.uniform(0, 1)
    energia_consumida =
        random.uniform(0, 1)
    temperatura_ambiente =
        random.uniform(0, 1)
    nivel_bateria = random.uniform(0,
        1)

    return velocidad_lineal,
        velocidad_angular,
        aceleracion_lineal,
        aceleracion_angular, tiempo, \
            energia_consumida,
                temperatura_ambiente,
                nivel_bateria

def generar_dataset(num_muestras,
    csv_filename="datos_robot.csv"):
    datos = []
    posicion_actual = np.array([0, 0])

    with open(csv_filename, 'w',
        newline='') as csvfile:
        fieldnames = ['x', 'y',
            'objeto_presente',
            'lectura_ultrasonico',
            'movimiento_direccion',
            'movimiento_distancia',
                        'velocidad_linea
                            l',
                            'velocidad_a
                            ngular',
                            'aceleracion
                            _lineal',
                            'aceleracion
                            _angular',
                            'tiempo',
                        'energia_consumi
                            da',
                            'temperatura
                            _ambiente',
                            'nivel_bater
                            ia']
        writer =
            csv.DictWriter(csvfile,
            fieldnames=fieldnames)

        writer.writeheader()

        for _ in range(num_muestras):
            movimiento =
                simular_movimiento()
            posicion_actual[0] += movi
                miento['distancia'] *
                np.cos(movimiento['dir
                eccion'])
            posicion_actual[1] += movi
                miento['distancia'] *
                np.sin(movimiento['dir
                eccion'])
```

```
            objeto_presente =
            ↪  random.choice([True,
            ↪  False])

            lectura_ultrasonico =
            ↪  simular_lectura_ultras
            ↪  onico(posicion_actual,
            ↪  objeto_presente)
            (velocidad_lineal,
            ↪  velocidad_angular,
            ↪  aceleracion_lineal,
            ↪  aceleracion_angular,
             tiempo,
              ↪  energia_consumida,
              ↪  temperatura_ambiente,
              ↪  nivel_bateria) =
              ↪  simular_otras_caracte
              ↪  risticas()
            datos.append({
                'x':
                 ↪  posicion_actual[0],
                'y':
                 ↪  posicion_actual[1],
                'objeto_presente':
                 ↪  objeto_presente,
                'lectura_ultrasonico':
                 ↪  lectura_ultrasonic
                 ↪  o,
                'movimiento_direccion'
                 ↪  :
                 ↪  movimiento['direcc
                 ↪  ion'],
                'movimiento_distancia'
                 ↪  :
                 ↪  movimiento['distan
                 ↪  cia'],
                'velocidad_lineal':
                 ↪  velocidad_lineal,
                'velocidad_angular':
                 ↪  velocidad_angular,
                'aceleracion_lineal':
                 ↪  aceleracion_lineal,
                'aceleracion_angular':
                 ↪  aceleracion_angula
                 ↪  r,
                'tiempo': tiempo,
                'energia_consumida':
                 ↪  energia_consumida,
                'temperatura_ambiente'
                 ↪  :
                 ↪  temperatura_ambien
                 ↪  te,
                'nivel_bateria':
                 ↪  nivel_bateria
            })

            writer.writerow({
                'x':
                 ↪  posicion_actual[0],
                'y':
                 ↪  posicion_actual[1],
                'objeto_presente':
                 ↪  objeto_presente,
                'lectura_ultrasonico':
                 ↪  lectura_ultrasonic
                 ↪  o,
                'movimiento_direccion'
                 ↪  :
                 ↪  movimiento['direcc
                 ↪  ion'],
                'movimiento_distancia'
                 ↪  :
                 ↪  movimiento['distan
                 ↪  cia'],
                'velocidad_lineal':
                 ↪  velocidad_lineal,
                'velocidad_angular':
                 ↪  velocidad_angular,
                'aceleracion_lineal':
                 ↪  aceleracion_lineal,
                'aceleracion_angular':
                 ↪  aceleracion_angula
                 ↪  r,
                'tiempo': tiempo,
                'energia_consumida':
                 ↪  energia_consumida,
                'temperatura_ambiente'
                 ↪  :
                 ↪  temperatura_ambien
                 ↪  te,
                'nivel_bateria':
                 ↪  nivel_bateria
            })

        print(f"El conjunto de datos ha
         ↪  sido guardado en
         ↪  '{csv_filename}'.")
        return datos

    num_observaciones = 2000
    conjunto_datos =
     ↪  generar_dataset(num_observaciones)
    for i, observacion in
     ↪  enumerate(conjunto_datos):
        print(f"Observacin {i+1}:
         ↪  {observacion}")
```

Appendix B
Supervised Learning - Data Processing and Model Analysis

```
import cv2
import numpy as np
```

```
from matplotlib import pyplot as plt
import seaborn as sns
```

```
df = pd.read_csv('datos_robot_copy.csv')
df
```

```
print("Valores nulos por columna:\n",
↪  df.isnull().sum())
```

```
print("Columnas con valores
↪  categóricos:\n",
↪  df.select_dtypes(include='object').
columns)
```

```
from sklearn.preprocessing import
↪  LabelEncoder

label_encoder = LabelEncoder()

df['objeto_presente_si']
↪  =label_encoder.fit_transform
(df['objeto_presente'])

df[['objeto_presente',
↪  'objeto_presente_si']].head()
```

```
print("Cantidad de valores igual a 1 en
↪  objeto_presente_si:",
↪  df['objeto_presente_si'].sum())
```

```
columnas_a_eliminar = ['objeto_presente',
↪  'energia_consumida',
↪  'aceleracion_angular',]
df = df.drop(columns=columnas_a_eliminar)
df.head()
```

```
y = df['objeto_presente_si']
X = df.drop(columns=
['objeto_presente_si'])
```

```
from sklearn.model_selection import
↪  train_test_split
from sklearn.svm import SVC
from sklearn.metrics import
↪  accuracy_score, confusion_matrix

X_train, X_test, y_train, y_test =
↪  train_test_split(X, y, test_size=0.2,
↪  random_state=42)

modelo_svm = SVC(kernel='linear', C=1.0)

modelo_svm.fit(X_train, y_train)

y_pred = modelo_svm.predict(X_test)

precision = accuracy_score(y_test, y_pred)
print(f"Precisión del modelo:
↪  {precision}")
```

```
matriz_confusion =
↪  confusion_matrix(y_test, y_pred)

fig, ax = plt.subplots(figsize=(8, 6))

class_names = ['No Detectado',
↪  'Detectado']

sns.heatmap(matriz_confusion, annot=True,
↪  fmt='d', cmap='Blues', cbar=False,
↪  ax=ax,
            xticklabels=class_names,
             ↪  yticklabels=class_names)

plt.title('Matriz de Confusión - Detección
↪  de Objetos')
plt.xlabel('Predicciones del Modelo')
plt.ylabel('Valores Reales')
plt.yticks(rotation=0)
plt.show()
```

```
plt.figure(figsize=(8, 8))
plt.scatter(X_test.loc[y_pred == 1, 'x'],
↪  X_test.loc[y_pred == 1, 'y'],
↪  marker='x', label='Obstáculo',
↪  color='red', zorder=2)
plt.plot(X_test['x'], X_test['y'],
↪  marker='o', label='Trayectoria del
↪  robot', color='blue', zorder=1)

plt.title('Trayectoria del Robot con
↪  Predicciones de Obstáculos')
plt.xlabel('Coordenada X')
plt.ylabel('Coordenada Y')
plt.legend()
plt.grid(True)
plt.show()
```

Appendix C
Unsupervised Learning - Simulated Data Generation

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.preprocessing import
↪  StandardScaler
from sklearn.decomposition import PCA

file_path =
↪  "/content/sensor_readings_24.data"
column_names = [f"Sensor_{i}" for i in
↪  range(24)] + ["Class Label"]
df = pd.read_csv(file_path, sep=",",
↪  header=None, names=column_names)
numerical_features = df.iloc[:,
↪  :-1].values
class_labels = df["Class Label"].values
scaler = StandardScaler()
scaled_numerical_features = scaler.fit
↪  _transform(numerical_features)
k = 4
kmeans = KMeans(n_clusters=k,
↪  random_state=42)
clusters = kmeans.fit_predict(scaled_n
↪  umerical_features)
```

```
pca = PCA(n_components=2)
reduced_numerical_features = pca.fit_t
↪ ransform(scaled_numerical_features)

plt.scatter(reduced_numerical_features
↪ [:, 0],
↪ reduced_numerical_features[:, 1],
↪ c=clusters, cmap='viridis',
↪ alpha=0.5)
plt.title('K-Means Clustering of
↪ Wall-Following Robot Navigation
↪ Data (24 Sensor Readings)')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.show()

unique_labels = np.unique(class_labels)
plt.figure()
for label in unique_labels:
    indices = np.where(class_labels ==
    ↪ label)
    plt.scatter(reduced_numerical_feat
    ↪ ures[indices, 0],
    ↪ reduced_numerical_features[ind
    ↪ ices, 1], label=label,
    ↪ alpha=0.5)
plt.title('Class Labels of
↪ Wall-Following Robot Navigation
↪ Data')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend()
plt.show()
```

Appendix D
Reinforcement Learning - Environment

```
import gym
from gym import spaces
import numpy as np

class FastestRouteFindingRobotEnv(gym.
↪ Env):
    def __init__(self, map_size=10):
        super(FastestRouteFindingRobot
        ↪ Env,
        ↪ self).__init__()

        self.observation_space =
        ↪ spaces.Discrete(map_size)
        self.action_space =
        ↪ spaces.Discrete(3)
        self.robot_position = 0
        self.goal_position = map_size
        ↪ - 1

    def reset(self):
        self.robot_position = 0
        return self.robot_position

    def step(self, action):
        if action == 0:
            self.robot_position = min(
            ↪ self.robot_position +
            ↪ 1, self.observation_sp
            ↪ ace.n -
            ↪ 1)
        elif action == 1:
            self.robot_position = max(
            ↪ self.robot_position -
            ↪ 1, 0)

        done = (self.robot_position ==
        ↪ self.goal_position)
        reward =
        ↪ -abs(self.robot_position -
        ↪ self.goal_position)

        done = done or
        ↪ (self.robot_position ==
        ↪ self.observation_space.n -
        ↪ 1)

        return self.robot_position,
        ↪ reward, done, {}
```

Appendix E
Reinforcement Learning - Training

```
import numpy as np

class FastestRouteFindingRobotEnv:
    def __init__(self, map_size):
        self.map_size = map_size
        self.robot_position = 0
        self.target_position =
        ↪ map_size - 1
        self.observation_space =
        ↪ np.arange(map_size)
        self.action_space =
        ↪ np.array([0, 1])

    def reset(self):
        self.robot_position = 0
```

```
        return self.robot_position

    def step(self, action):
        if action == 0:
            self.robot_position = min(self.robot_position + 1, self.map_size - 1)
        elif action == 1:
            self.robot_position = max(self.robot_position - 1, 0)

        done = self.robot_position == self.target_position

        distance_penalty = -0.1 * abs(self.robot_position - self.target_position)
        reward = 1 if done else 0
        reward += distance_penalty

        return self.robot_position, reward, done, {}

class QLearningAgent:
    def __init__(self, state_space_size, action_space_size, learning_rate=0.01, discount_factor=0.95, exploration_prob=0.5, exploration_decay=0.001):
        self.state_space_size = state_space_size
        self.action_space_size = action_space_size
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.exploration_prob = exploration_prob
        self.exploration_decay = exploration_decay

        self.q_table = np.zeros((state_space_size, action_space_size))

    def choose_action(self, state):
        if np.random.rand() < self.exploration_prob:
            return np.random.choice(self.action_space_size)
        else:
            return np.argmax(self.q_table[state, :])

    def update_q_table(self, state, action, reward, next_state):
        self.q_table[state, action] += self.learning_rate * (reward + self.discount_factor * np.max(self.q_table[next_state, :]) - self.q_table[state, action])

    def decay_exploration_prob(self):
        self.exploration_prob = max(0.1, self.exploration_prob - self.exploration_decay)
env = FastestRouteFindingRobotEnv(map_size=15)
agent = QLearningAgent(state_space_size=env.observation_space.size, action_space_size=env.action_space.size)
num_episodes = 1000

for episode in range(num_episodes):
    state = env.reset()
    done = False

    while not done:
        action = agent.choose_action(state)
        next_state, reward, done, _ = env.step(action)
        agent.update_q_table(state, action, reward, next_state)

        state = next_state

    agent.decay_exploration_prob()

    if episode % 100 == 0:
        print(f"Episode {episode}, Exploration Probability: {agent.exploration_prob}, Steps: {env.robot_position}, Reward: {reward}")
```

APPENDIX F
REINFORCEMENT LEARNING - AGENT

```
import numpy as np

class FastestRouteFindingRobotEnv:
    def __init__(self, map_size):
        self.map_size = map_size
        self.robot_position = 0
        self.target_position =
        ↪  map_size - 1
        self.observation_space =
        ↪  np.arange(map_size)
        self.action_space =
        ↪  np.array([0, 1])

    def reset(self):
        self.robot_position = 0
        return self.robot_position

    def step(self, action):
        if action == 0:  # Move right
            self.robot_position = min(
            ↪  self.robot_position +
            ↪  1, self.map_size - 1)
        elif action == 1:  # Move left
            self.robot_position = max(
            ↪  self.robot_position -
            ↪  1, 0)

        done = self.robot_position ==
        ↪  self.target_position

        # Penalize the distance to the
        ↪  target
        distance_penalty = -0.1 *
        ↪  abs(self.robot_position -
        ↪  self.target_position)

        # Reward for reaching the
        ↪  target
        reward = 1 if done else 0

        reward += distance_penalty

        return self.robot_position,
        ↪  reward, done, {}

class QLearningAgent:
    def __init__(self,
    ↪  state_space_size,
    ↪  action_space_size,
    ↪  learning_rate=0.01,
    ↪  discount_factor=0.95,
    ↪  exploration_prob=0.5,
    ↪  exploration_decay=0.001):
        self.state_space_size =
        ↪  state_space_size
        self.action_space_size =
        ↪  action_space_size
        self.learning_rate =
        ↪  learning_rate
        self.discount_factor =
        ↪  discount_factor
        self.exploration_prob =
        ↪  exploration_prob
        self.exploration_decay =
        ↪  exploration_decay

        self.q_table = np.zeros((state
        ↪  _space_size,
        ↪  action_space_size))

    def choose_action(self, state):
        if np.random.rand() <
        ↪  self.exploration_prob:
            return np.random.choice(se
            ↪  lf.action_space_size)
        else:
            return np.argmax(self.q_ta
            ↪  ble[state,
            ↪  :])

    def update_q_table(self, state,
    ↪  action, reward, next_state):
        self.q_table[state, action] +=
        ↪  self.learning_rate * (
                reward + self.discount
                ↪  _factor *
                ↪  np.max(self.q_tabl
                ↪  e[next_state, :])
                ↪  - self.q_table[sta
                ↪  te,
                ↪  action]
        )

    def decay_exploration_prob(self):
        self.exploration_prob =
        ↪  max(0.1,
        ↪  self.exploration_prob -
        ↪  self.exploration_decay)

env = FastestRouteFindingRobotEnv(map_
↪  size=15)
agent = QLearningAgent(state_space_siz
↪  e=env.observation_space.size,
↪  action_space_size=env.action_space
↪  .size)

num_episodes = 1000

for episode in range(num_episodes):
    state = env.reset()
```

```
    done = False

    while not done:
        action =
        ↪  agent.choose_action(state)
        next_state, reward, done, _ =
        ↪  env.step(action)
        agent.update_q_table(state,
        ↪  action, reward, next_state)

        state = next_state

    agent.decay_exploration_prob()

    if episode % 100 == 0:
        print(f"Episode {episode},
        ↪  Exploration Probability:
        ↪  {agent.exploration_prob},
        ↪  Steps:
        ↪  {env.robot_position},
        ↪  Reward: {reward}")
```